A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs

Gwenaël Delaval
INRIA-IRISA
Rennes
France
Gwenael.Delaval@inria.fr

Alain Girault
INRIA Rhône-Alpes
Grenoble
France
Alain.Girault@inria.fr

Marc Pouzet
LRI, Univ. Paris-Sud
Orsay
France
Marc.Pouzet@lri.fr

Abstract

We address the design of distributed systems with synchronous dataflow programming languages. As modular design entails handling both architectural and functional modularity, our first contribution is to extend an existing synchronous dataflow programming language with primitives allowing the description of a distributed architecture and the localization of some expressions onto some processors. We also present a distributed semantics to formalize the distributed execution of synchronous programs. Our second contribution is to provide a type system, in order to infer the localization of non-annotated values by means of type inference and to ensure, at compilation time, the consistency of the distribution. Our third contribution is to provide a type-directed projection operation to obtain automatically, from a centralized typed program, the local program to be executed by each computing resource. The type system as well as the automatic distribution mechanism has been fully implemented in the compiler of an existing synchronous data-flow programming language.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.2 [*Programming Languages*]: Languages Classifications—Concurrent, distributed, and parallel languages

General Terms Languages

Keywords Synchronous programming, distribution, type systems, functional programming

1. Motivations

Synchronous programming languages [5] are frequently used in the industry for the design of real-time embedded systems. Such languages define deterministic behaviors and lie on formal semantics, making them suitable for the design and implementation of safety critical systems. They are used, for example, in critical domains such as the automotive, avionics, or nuclear industry.

Most of the systems designed with synchronous languages are centralized systems. The parallelism expressed in these languages is a *functional* one, whose purpose is to ease the design process by providing ideal timing and concurrency constructs to the designers. A synchronous program is then compiled into a sequential program emulating the parallel execution of the functional parallel branches. This sequential program is executed on a single computing resource. Yet, most embedded systems are composed of several computing resources (named "locations"), for reasons such as performance, dedicated actuators or sensors drivers, or adaptivity of the locations to the tasks they are assigned to (e.g., pure computing tasks vs control tasks). We call this the *execution* parallelism. This paper addresses the problem of mapping the functional parallelism onto the execution one, in a modular way. We focus on distributed systems implemented as networks of deterministic processes communicating with FIFOs.

Fragments of a distributed system can be designed separately; but in complex and multifunctional embedded systems, functionalities are frequently independent of the hardware architecture, implying conflicts between architectural and functional modularity. Thus, one functionality can use several locations and one location can be involved in several functionalities. As a result, programming each location separately compromises the modularity and is error-prone. This situation occurs within several industrial areas, such as automotive embedded systems, and software-defined radio [15].

Our paper is organized as follows: Section 2 gives an overview of the context, and motivates our method through some examples and one application. Section 3 presents the semantics and the formalization of the spatial type system. Section 4 presents the projection operation, which is our third contribution. Finally, related work and discussion about the solution will be exposed in Section 5.

2. Overview

2.1 Distribution of Synchronous Dataflow Programs

Synchronous dataflow languages, such as Lustre, Signal, or Lucid Synchrone [5], manipulate infinite streams of values as primitive values: the notation `1` represents the infinite stream $1, 1, ...$, while `int` stands for the type of infinite streams of integers. For any stream x, we note x_i its ith value. In this context, functions (called nodes hereafter) are stream functions: e.g., $\texttt{int} \rightarrow \texttt{int}$ is the type of functions from integer streams to integer streams. Combinatorial functions are implicitly lifted to apply pointwise to their arguments: e.g., if $\texttt{x} = (x_i)_{i \in I\!N}$ and $\texttt{y} = (y_i)_{i \in I\!N}$ are two integer streams, then $(\texttt{x} + \texttt{y}) = (x_i + y_i)_{i \in I\!N}$. Moreover, we use a unitary delay, noted `fby`, such that $(\texttt{x fby y})_i = x_0$ if $i = 0$ and y_{i-1} otherwise.

Such a program is classically compiled into a single function f, which computes the values of outputs and updates the system's state, from the values of inputs and the current state. This function f is then embedded inside a periodic execution loop. Our contribution is to extend this classical compilation scheme to a distributed framework: the result of the compilation of a distributed system will consist of n functions f_i, one for each location i, which will compute the values of outputs, communication channels, and lo-

LCTES'08, June 12–13, 2008, Tucson, Arizona, USA.

cal state, from the values of inputs, other incoming communication channels, and the current local state.

2.2 Language-based Distribution

We address *functional* distribution, not achieved for the sake of performance but because the system is intrinsically distributed. Distribution is driven by the fact that some functions have a meaning only at some specific locations and not at others. We can think, e.g., of a function returning the value of a physical sensor and which has to be executed where the sensor is. Therefore, locations will be defined by the functionalities they provide.

Designing such distributed systems is non-trivial, because of problems such as the scheduling of communications or the type consistency of the communicated data. The usual method, using architecture languages like AADL [2], involves describing the system's architecture by partitioning it in subsystems. Each subsystem can then be defined separately, possibly with different languages. However, in the case of tightly dependent subsystems, where conflicts between architectural and functional modularity can occur, it is less error-prone and more efficient to define the system as a whole, together with architectural annotations. Our first contribution is to provide language primitives to allow the programmer to describe the architecture, and to express where some values are located, i.e., on which location some computations are performed.

The architecture is described by the explicit declaration of the set of existing locations and the links between them. At this point, locations are symbolic: a location declaration introduces a symbolic name, which will then be used to express the fact that a stream is computed or available at this symbolic location. We define in Section 4.3 a *projection* operation which produces, for each symbolic name, a single non-distributed synchronous program to be executed at the physical location represented by this symbolic name.

The syntax for declaring the physical location `A` is `loc A`. The existence of a communication link from `A` to `B` is declared by `link A to B`. Note that we distinguish *communication links* from *communication channels*, introduced in Section 3.4: communication links, specified by the primitive `link`, state the *ability* to communicate from one location to another. In contrast, actual *channels* used by the distributed system are inferred by the type system.

The statement e `at` A means that every value used in the expression e (streams and nodes composing its subterms) will be located at A. The programmer does not need to express the localization of every value. Our second contribution is to provide a type and effects system [19] whose double function is to check the validity of the localization expressed w.r.t. the architecture, and to infer the localization of non-explicitly located values. For instance, the node `f` given below consists of two computations `g` and `h`, respectively located by the programmer on locations `A` and `B`, thanks to the `at A` and `at B` annotations.

```
node f(x) = z with
       y = g(x) at A
  and z = h(y) at B;
```

Communications are abstracted, and thus not expressed by the programmer, neither technically, nor concerning their place inside the code. The technical expression of communications is left to the further phase of integration on actual architecture: our method only deals with inferring the localization of these communications, and their coherence throughout the distributed code. We assume for now that communications can occur at any localization, and can concern any value entirely concealed within a location (i.e., not the distributed data structures, like distributed pairs). From a programmer's point of view, this choice is a compromise between no control at all (communications are possible everywhere) and absolute control (the programmer expresses every communication).

2.3 A Spatial Type System for Automatic Distribution

We place ourselves in a functional framework, where for the sake of modularity, functions can neither be inlined nor analyzed dependently of their calling context. We provide a special type system dedicated to the distributed execution of the program, as an analysis provided to the programmer to help him ensure the consistency of the distribution specification. We call it a *spatial type system* and, when clear from context, we shall simply refer to it as a type system. This spatial type system describes the localization of streams, and a type-directed approach is followed to achieve code distribution. This also allows us to preserve higher-order features, hence allowing the expression of dynamic reconfiguration of nodes by application of other nodes as inputs.

The other motivation for using a type system is to achieve type inference: in order not to force the programmer to specify everything (i.e., the localization of each stream), spatial types will be inferred from the available spatial annotations in the source. The spatial type system also checks the consistency of these annotations with the given architecture. Spatial consistency means, e.g., that applying a node located on a location to a stream located elsewhere is not correct. As we are in a functional context, spatial types will be inferred for each defined node modularly.

A typed program is then automatically distributed by the compiler, by extracting, for each declared location, one program strictly composed of computations to be performed on this location, as well as added communications from and to other locations in the form of added inputs and outputs.

The spatial type of a stream is the location where this stream is located. In the case of a stream whose values are communicated via a channel from one location to another, its spatial type is a set: it is the set of locations where the stream will be available. The spatial type of a node f is written $t_i \multimap\!\langle S \rangle\!\rightarrow t_o$, where t_i and t_o are respectively the spatial types of f's inputs and outputs, and S is the set of locations involved in the computation of f. This set of locations can be larger than the union of t_i and t_o's sets of locations, since the computation of f can involve intermediary locations.

2.4 Examples

All the examples below assume the architecture declaration:

```
loc A;  loc B;  link A to B;
```

The first example is a sequence of three nodes `f1`, `f2` and `f3`, each assumed to be of spatial type $\forall \delta.c$ `at` $\delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c$ `at` δ. `f1` and `f3` are localized by the programmer, respectively on `A` and `B`. `f2` is not explicitly localized.

```
node g(x) = y3 with
       y1 = f1(x) at A
  and y2 = f2(y1)
  and y3 = f3(y2) at B
```

This node will be given the spatial type c `at` $A \multimap\!\langle\{A, B\}\rangle\!\rightarrow c$ `at` B. As the localization of computations has to be done modularly, a spatial type for `f2` will be given once, among the two possibilities c `at` $A \multimap\!\langle\{A\}\rangle\!\rightarrow c$ `at` A and c `at` $B \multimap\!\langle\{B\}\rangle\!\rightarrow c$ `at` B. In contrast, since there is no communication link from `B` to `A`, the following node will be rejected by the type system:

```
node g'(x) = y3 with
       y1 = f1(x) at B
  and y2 = f2(y1)
  and y3 = f3(y2) at A
```

Furthermore, it can be noted here that node `g` cannot be used within a located declaration. The following node will be rejected by our type system:

```
node g'(m,x) = y with
  y = g(m,x) at A
```

The second example involves a higher-order node: the node `h` takes as input two nodes `f` and `g`, and an input `x`, and applies `f` to `x` at one location, and then `g` to the result of the first application at another location. This example shows also how a node can be defined with local locations for more modularity. These new locations are introduced as a list between $[\ldots]$, can then be used within the node. This higher-order node uses two location variables δ_1 and δ_2:

```
node h [δ1, δ2] (f,g,x) = z with
       y = f(x) at δ1
   and z = g(y) at δ2
```

`h` receives then the spatial type:

$$\forall \alpha, \beta, \gamma. \forall \delta_1, \delta_2 : \{\delta_1 \triangleright \delta_2\}.$$
$$\Big(\quad (\alpha \texttt{ at } \delta_1 \multimap\!\langle\{\delta_1\}\rangle\!\rightarrow \beta \texttt{ at } \delta_1)$$
$$\times(\beta \texttt{ at } \delta_2 \multimap\!\langle\{\delta_2\}\rangle\!\rightarrow \gamma \texttt{ at } \delta_2)$$
$$\times(\alpha \texttt{ at } \delta_1)\Big) \multimap\!\langle\{\delta_1, \delta_2\}\rangle\!\rightarrow \gamma \texttt{ at } \delta_2$$

The set of constraints ($\{\delta_1 \triangleright \delta_2\}$) is inferred from the links required by the node. These constraints are resolved, with the actual architecture, when this node is instantiated. A constraint $\delta \triangleright \delta'$ is resolved, either by stating $\delta = \delta' = s$, or with two locations s and s' such that there exists a communication link from s to s' in the local architecture. Thus, the node `h` can be instantiated in these two ways (assuming the existence of two nodes `f` and `g`, both of spatial type $\forall \delta. c \texttt{ at } \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c \texttt{ at } \delta$):

```
    y1 = h (f at A,g at A,x1)
and y2 = h (f at A,g at B,x2)
```

We can observe than an arrow type appearing on the left of another arrow type cannot comprise more than one location. This is caused by the form taken by the distribution: since the projection operation on one node is performed on locations, and not sets of locations, we cannot handle effect variables, unlike other type and effect systems.

2.5 Application

As a concrete example, we consider the definition of a reception channel of a software radio. A *software radio*, or *software-defined radio*, is a radio in which components usually defined as hardware, e.g., demodulation or filter components, are defined as software [15]. This allows in particular the reconfiguration, possibly dynamic, of these components.

Consider a reception channel composed of three main components: a pass-band filter allowing the selection of the carrier wave, a demodulator component, and a component allowing the analysis of the received signal, e.g., an error-correction function. For the sake of performance, these components are usually implemented on different architectural elements: the pass-band filter on a FPGA, the demodulator on a digital signal processor (DSP), and the error correction on a general-purpose processor (GPP).

Each component of this reception channel could easily be defined separately. But in the case of software-defined radio, the system must support several functionalities [10]: each of these functionalities must be engineered separately, and then integrated together. Then, there is a conflict between distribution and functional modularity issues.

Let us study the case of a multichannel reception system that supports the two mobile standards GSM and UMTS: the former involves a filter for 1800 MHz frequencies, a GMSK demodulator, and a CRC / convolutional error correction module, while the latter involves a 2 GHz filter, a QPSK demodulator, and a CRC / convolutional / turbo codes error correction module.

Figure 1 shows an implementation of this reception channel on a system composed of three hardware components: a FPGA dedicated to the execution of the two pass-band filters, a DSP for the demodulation functions, and a GPP for error-correction modules and for the control of the whole system, i.e., in this case, the switch between the two channels. This system has one input `x`, the radio signal from the antenna. `y` denotes the output signal of the system, i.e., the decoded and corrected information received by the channel. From this value, a function `gsm_or_umts` (noted `g` on Figure 1 for the sake of brevity), local at GPP, computes what channel will be used at next instant. In this figure, each location is graphically represented by a gray box.

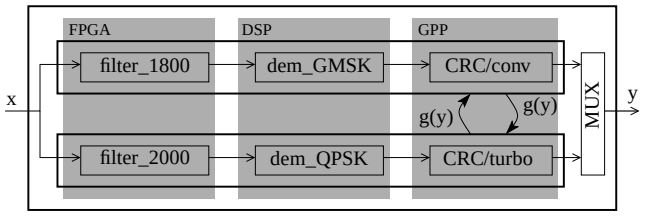

Figure 1. Functional model of a multichannel software radio.

In a classical context, designing this multichannel software radio would be performed by separately programming each of the three hardware components, which raises two problems. Firstly, there is no guarantee that the components interact as specified: i.e., the 1800 filter with the GMSK demodulator, and so on. This requires the MUX function to be duplicated on the three computing resources, so as to guarantee the correctness of the system. This situation compromises the modularity of the system. Secondly, each of the two channels corresponds to an independent software entity. Programming independently each hardware component leads to the separate design, at least from some point of the design flow, of closely related software components (e.g., filter and demodulator of the same channel).

For the sake of modularity, this system would be better designed by considering the channels independently, and not the hardware components. This situation suggests adding primitives allowing to express the localization of streams directly in the language. Such primitives should allow the programming of software components independently of the architecture, handled as a separate concern. Thus, consistency analysis such as data typing could be performed on the global program: communication channels could be typed and the data consistency of the whole system could be checked. This way, inconsistencies due to serialisation could be detected at compilation time.

The code below implements this multichannel software radio, with our extended language. The architecture consists of three locations, `FPGA`, `DSP`, and `GPP`, completely connected:

```
node channel(filter,demod,crc,x) = y with
      f = filter(x) at FPGA
  and d = demod(f) at DSP
  and y = crc(d) at GPP

node multichannel_sdr(x) = y with
      c = gsm_or_umts(y) at GPP in
  and
    if (true fby c) then
      y = channel(filter_1800,gmsk,conv,x)
    else
      y = channel(filter_2000,qpsk,turbo,x)
```

This implementation strictly follows the architecture of the system described in Figure 1. It shows the declaration of three symbolic locations (`FPGA`, `DSP`, and `GPP`). We assume that all filter, demodulation, and correction functions are local ones, i.e., they are of

spatial type $\forall\delta.c$ `at` $\delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c$ `at` δ. Since the conditional construct comprises declarations that have to be executed on the set of locations $\{\texttt{FPGA}, \texttt{DSP}, \texttt{GPP}\}$, `c` is thus inferred to be communicated to these locations. The conditional `if/then/else` is evaluated with the value of `c` at the previous instant. The distribution of this example will put a copy of this `if/then/else` on these three locations. Finally, the expression `true fby c` will be computed at `GPP`, since the result of this expression has to be communicated to the three locations where the conditional construct will be duplicated.

By the same reasoning, we can infer that the spatial type of `x` is `FPGA`, and the one of `y` is `GPP`. As a result, the spatial type of the node `multichannel_sdr` is:

$$(c \texttt{ at FPGA}) \multimap\!\langle\{\texttt{FPGA}, \texttt{DSP}, \texttt{GPP}\}\rangle\!\rightarrow (c \texttt{ at GPP})$$

3. Formalization

We first define a synchronous dataflow core language (Section 3.1), and give its centralized semantics (Section 3.2) and its distributed semantics (Section 3.3). The centralized semantics is considered to be the reference semantics and we only consider programs that react w.r.t. this semantics. Programs that do not react (e.g., for typing or causality reasons) are assumed to be rejected by other means [9]. The distributed semantics allows us to give a meaning to location annotations. A spatial type system is then presented (Section 3.4). It is used to both reject programs which cannot be distributed and to annotate every expression from the source code with explicit locations.

3.1 The Core Language Syntax

$$\begin{array}{lcl} P & ::= & \mathcal{A};d;D \\ \mathcal{A} & ::= & \mathcal{A};\mathcal{A} \mid \texttt{loc } A \mid \texttt{link } A \texttt{ to } A \\ s & ::= & \delta \mid A \\ d & ::= & \texttt{node } f[\delta_1,\ldots,\delta_n](p) = e \texttt{ with } D \mid d;d \\ D & ::= & p = e \mid p = x(e) \mid D \texttt{ and } D \mid \texttt{if } e \texttt{ then } D \texttt{ else } D \\ p & ::= & p,p \mid x \\ e & ::= & i \mid x \mid (e,e) \mid \mathrm{op}(e,e) \mid e \texttt{ fby } e \mid e \texttt{ at } s \end{array}$$

A program is made of an architecture description ($\mathcal{A}$), a sequence of node definitions (d) and a main set of equations (D). An architecture description is a set of declarations of locations (`loc` A) or links (A `link to` A) which state the existence of a communication link from one location to another. A location s is either a location variable δ, or a location constant A. A node definition is composed of an expression and a set of equations. A set of local locations $\{\delta_1,\ldots,\delta_n\}$ can be associated as location parameters of a node definition (`node` $f[\delta_1,\ldots,\delta_n](p) = e$ `with` D). Definitions D define patterns of variables p, and are either single equations ($p = e$), definitions naming the result of an application ($p = x(e)$), parallel declarations (D `and` D), or alternative declarations (`if` e `then` D `else` D). An expression e may be an immediate value (i), a variable (x), a pair construction (e, e ; pair destruction can be performed by pattern definitions), a binary combinatorial operation ($\mathrm{op}(e,e)$, where op can be $(+)$, $(-)$,...), an initialized delay (e `fby` e), or an expression annotated with an explicit location s (e `at` s).

3.2 The Centralized Synchronous Semantics

The purpose of the centralized semantics is to serve as a reference semantics. This semantics does not take into account distribution primitives. We first introduce auxiliary definitions. A value is either an immediate constant (i), a pair or a function.

$$\begin{array}{l} v ::= i \mid (v,v) \mid \lambda x.e \texttt{ with } D \\ R ::= [v_1/x_1,\ldots,v_n/x_n] \text{ s.t. } \forall i \neq j, x_i \neq x_j \\ S ::= R_1.R_2\ldots \end{array}$$

A reaction environment R associates values to names and assumes that names are pairwise distinct. S denotes a sequence of reaction environments.

Given a sequence d of node definitions `node` $f_i[\vec{\delta_i}](x_i) = e_i$ `with` D_i, an initial global environment R_d is defined, holding λ-values of each f_i. This initial environment will be given as input of the main program.

$$R_d = [\lambda x_1.e_1 \texttt{ with } D_1/f_1,\ldots,\lambda x_n.e_n \texttt{ with } D_n/f_n]$$

The synchronous centralized semantics is defined by means of two reaction predicates. $R \vdash e_1 \xrightarrow{v} e_2$ states that in the reaction environment R, the expression e_1 emits the value v and rewrites into the new expression e_2. The predicate $R \vdash D_1 \xrightarrow{R'} D_2$ states that in the reaction environment R, the declaration D_1 defines the reaction environment R' and rewrites into D_2. The centralized execution of a program P is denoted $S_i \vdash P : S_o$, meaning that under a sequence of input environments $S_i = R_1.R_2\ldots$, the program $P = \mathcal{A};d;D$ produces a sequence of output environments $S_o = R'_1.R'_2\ldots$ such that:

$$\frac{R_d, R_i, R_o \vdash D \xrightarrow{R_o} D' \qquad S_i \vdash \mathcal{A};d;D' : S_o}{R_i.S_i \vdash \mathcal{A};d;D : R_o.S_o}$$

The rules for the reaction predicates are given in Figure 2.

$$\text{(IMM) } R \vdash i \xrightarrow{i} i \qquad\qquad \text{(INST) } R, [v/x] \vdash x \xrightarrow{v} x$$

$$\text{(FBY) } \frac{R \vdash e_1 \xrightarrow{v_1} e'_1 \qquad R \vdash e_2 \xrightarrow{v_2} e'_2}{R \vdash e_1 \texttt{ fby } e_2 \xrightarrow{v_1} v_2 \texttt{ fby } e'_2}$$

$$\text{(OP) } \frac{R \vdash e_1 \xrightarrow{i_1} e'_1 \qquad R \vdash e_2 \xrightarrow{i_2} e'_2 \qquad i = \mathrm{op}(i_1,i_2)}{R \vdash \mathrm{op}(e_1,e_2) \xrightarrow{i} \mathrm{op}(e'_1,e'_2)}$$

$$\text{(PAIR) } \frac{R \vdash e_1 \xrightarrow{v_1} e'_1 \quad R \vdash e_2 \xrightarrow{v_2} e'_2}{R \vdash (e_1,e_2) \xrightarrow{(v_1,v_2)} (e'_1,e'_2)} \qquad \text{(AT) } \frac{R \vdash e \xrightarrow{v'} e'}{R \vdash e \texttt{ at } s \xrightarrow{v} e' \texttt{ at } s}$$

$$\text{(DEF) } \frac{R \vdash e \xrightarrow{(v_1,\ldots,v_n)} e'}{R \vdash (x_1,\ldots,x_n) = e \xrightarrow{[v_i/x_i]} (x_1,\ldots,x_n) = e'}$$

$$\text{(APP) } \frac{R(f) = \lambda p.e \texttt{ with } D \qquad R \vdash p' = e \texttt{ and } p = e' \texttt{ and } D \xrightarrow{R'} D'}{R \vdash p' = f(e') \xrightarrow{R'} D'}$$

$$\text{(AND) } \frac{R, R_2 \vdash D_1 \xrightarrow{R_1} D'_1 \qquad R, R_1 \vdash D_2 \xrightarrow{R_2} D'_2}{R \vdash D_1 \texttt{ and } D_2 \xrightarrow{R_1,R_2} D'_1 \texttt{ and } D'_2}$$

$$\text{(IF-1) } \frac{R \vdash e \xrightarrow{\texttt{true}} e' \qquad R \vdash D_1 \xrightarrow{R'} D'_1}{R \vdash \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{R'} \texttt{if } e' \texttt{ then } D'_1 \texttt{ else } D_2}$$

$$\text{(IF-2) } \frac{R \vdash e \xrightarrow{\texttt{false}} e' \qquad R \vdash D_2 \xrightarrow{R'} D'_2}{R \vdash \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{R'} \texttt{if } e' \texttt{ then } D_1 \texttt{ else } D'_2}$$

Figure 2. Centralized synchronous semantics.

An immediate value emits itself and rewrites to itself (rule IMM). A variable emits its current value as it is present in the reaction environment (rule INST). An initialized delay e_1 `fby` e_2 emits the first value of e_1, then the previous value of e_2 (rule FBY). An

operation is performed pointwisely on immediate values (rule OP). Pair construction follow classical rules (rule PAIR). Locations are not taken into account here (rule AT): annotations added by the programmer do not alter the centralized semantics of the program (i.e., its functionality). An equation $p = e$ emits the reaction environment defining the variables contained in p (rule DEF). A sequential function application is replaced by its body and argument definition (rule APP). Parallel equations are mutually recursive (rule AND). A conditional statement executes its first branch if its condition is true (rule IF-1) and its second branch otherwise (rule IF-2).

3.3 The Distributed Synchronous Semantics

The distributed semantics also operates on a program $P = \mathcal{A}; d; D$, but takes into account the architecture description and the explicit locations. However, it remains a synchronous semantics in the sense that the desynchronization due to the insertion of communications is not accounted for. It defines a *spatialized execution*: the values $\hat{v}$ emitted by expressions are now *distributed values*, i.e., they are annotated with location information stating how these values are distributed on the architecture:

$$\begin{array}{rcl} \hat{v} & ::= & lv \texttt{ at } A \mid (\hat{v}, \hat{v}) \mid \Lambda\vec{\delta}.\lambda x.e \texttt{ with } D \\ lv & ::= & i \mid (lv, lv) \\ \hat{R} & ::= & [\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n] \text{ s.t. } \forall i \neq j, x_i \neq x_j \\ \hat{S} & ::= & \hat{R}_1.\hat{R}_2.\ldots. \\ G & ::= & \langle \mathcal{S}, \mathcal{L} \rangle \end{array}$$

A distributed value $\hat{v}$ is either a local value lv, localized on the site A (lv `at` A), a distributed pair $(\hat{v}, \hat{v})$, or a node. A sequence of location parameters $\vec{\delta}$ is associated to nodes. A local value is either an immediate value i, or a local pair (lv, lv). $\hat{R}$ denotes a distributed reaction environment, and $\hat{S}$ a sequence of distributed environments. G denotes an architectural graphs composed of a set of locations $\mathcal{S}$, and a set of communcation links $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$.

Several values can represent the same distributed value:

$$(lv_1, lv_2) \texttt{ at } A = (lv_1 \texttt{ at } A, lv_2 \texttt{ at } A)$$

$$\frac{\hat{v}_1 = \hat{v}_1' \quad \hat{v}_2 = \hat{v}_2'}{(\hat{v}_1, \hat{v}_2) = (\hat{v}_1', \hat{v}_2')}$$

These equalities mean that a local pair $(1, 2)$, localized on the site A, can indifferently be denoted by the distributed values $(1, 2)$ `at` A or $(1 \texttt{ at } A, 2 \texttt{ at } A)$. A distributed pair can be, for example, the pair $(1 \texttt{ at } A, 2 \texttt{ at } B)$: the first compound is located on A, and the second on B.

The operator $\mathrm{loc}(\cdot)$ gathers the set of locations from a distributed value:

$$\begin{array}{lcl} \mathrm{loc}(i \texttt{ at } s) & = & \{s\} \\ \mathrm{loc}((\hat{v}_1, \hat{v}_2) \texttt{ at } s) & = & \mathrm{loc}(\hat{v}_1) \cup \mathrm{loc}(\hat{v}_2) \end{array}$$

The operator $|\cdot|$ erases annotations from a distributed value to get a centralized value, and extends straightforwardly to reaction environments:

$$|i \texttt{ at } s| = i \qquad |(\hat{v}_1, \hat{v}_2) \texttt{ at } s| = (|\hat{v}_1|, |\hat{v}_2|)$$

The distributed semantics is defined by means of two predicates refined from their centralized versions. $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e_2$ states that in the distributed reaction environment $\hat{R}$, the expression e_1 emits the distributed value $\hat{v}$ and rewrites into e_2. ℓ represents the set of locations involved in the computation of $\hat{v}$. The predicate for declarations is defined as well.

We denote by $\mathcal{S}$ the set of declared constant locations, and by $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ the set of declared communication links. The relation $\mathcal{L}$ defines the *ability* to communicate, and not the actual existence of communication channels, which will be inferred by the refined version of the type system. G denotes an architecture graph, composed of a set of locations $\mathcal{S}$, and a set of links $\mathcal{L}$ between these locations. An architecture description $\mathcal{A}$ defines an architecture graph G: the notation $G \vdash \mathcal{A} : G'$ means that given the architecture graph G, $\mathcal{A}$ defines the new architecture graph G'. The rules ARCH, DEF-LOC and DEF-LINK define this predicate:

$$\text{(ARCH)}\ \frac{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \mathcal{A}_1 : \langle \mathcal{S}_1, \mathcal{L}_1 \rangle \qquad \langle \mathcal{S}_1, \mathcal{L}_1 \rangle \vdash \mathcal{A}_2 : \langle \mathcal{S}_2, \mathcal{L}_2 \rangle}{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \mathcal{A}_1 ; \mathcal{A}_2 : \langle \mathcal{S}_2, \mathcal{L}_2 \rangle}$$

$$\text{(DEF-LOC)}\ \langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{loc } A : \langle \mathcal{S} \cup \{A\}, \mathcal{L} \rangle$$

$$\text{(DEF-LINK)}\ \frac{A_1, A_2 \in \mathcal{S}}{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{link } A_1 \texttt{ to } A_2 : \langle \mathcal{S}, \mathcal{L} \cup \{A_1 \mapsto A_2\} \rangle}$$

For clarity, we assume that the architecture graph G is global for subsequent semantic rules. The annotated execution of a program P is $\hat{S}_i \Vdash P : \hat{S}_o$, meaning that under a sequence of input environments $\hat{S}_i = \hat{R}_1.\hat{R}_2 \ldots$, the program $P = \mathcal{A}; d; D$ produces a sequence of output environments $\hat{S}_o = \hat{R}_1'.\hat{R}_2' \ldots$:

$$\frac{\langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : G \quad \hat{R}_d, \hat{R}_i, \hat{R}_o \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}_o} D' \quad \hat{S}_i \Vdash \mathcal{A}; d; D' : \hat{S}_o}{\hat{R}_i.\hat{S}_i \Vdash \mathcal{A}; d; D : \hat{R}_o.\hat{S}_o}$$

where $\hat{R}_d$ is defined from the sequence of node definitions $d = \texttt{node } f_i[\vec{\delta}_i](x_i) = e_i \texttt{ with } D_i$ as:

$$\hat{R}_d = [\Lambda\vec{\delta}_1.\lambda x_1.e_1 \texttt{ with } D_1/f_1, \ldots, \Lambda\vec{\delta}_n.\lambda x_n.e_n \texttt{ with } D_n/f_n]$$

The rules for the predicates $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e_2$ and $\hat{R} \overset{\ell}{\Vdash} D_1 \xrightarrow{\hat{R}'} D_2$ are given in Figure 3. An immediate value can be emitted anywhere (rule IMM). Rule INST defines the instantiation. A distributed value can be communicated from location s to location s' if there exists a communication link from s to s' (rule COMM). A binary operation can be performed only on immediate values located on the same location A; the result is located on A as well (rule OP). An annotated expression must involve at most the location stated for its computation (rule AT). An application involves choosing a set of constant locations, and replacing location parameters by these locations in the expression and the declaration (rule APP). The other rules state that the computation of a statement involves the union of the locations involved for the computation of its compounds.

Lemma 1 states that if a program reacts with the distributed semantics, then it reacts with the centralized one and produces the same values.

Lemma 1. *For all $D, D', \hat{R}, \hat{R}'$, if $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$, then there exists R, R' such that $R = |\hat{R}|$, $R' = |\hat{R}'|$ and $R \vdash D \xrightarrow{R'} D'$.*

3.4 Spatial Types

For the sake of clarity, we first present a simplified version of the type system. For projection, we refine this first version to take communication channels into account (Section 4).

The syntax of spatial type expressions is:

$$\begin{array}{rcl} \sigma & ::= & \forall\alpha_1, \ldots, \alpha_n.\forall\delta_1, \ldots, \delta_n : C.t \\ t & ::= & t \multimap\!\langle \ell \rangle\!\mapsto t \mid t \times t \mid tc \texttt{ at } s \\ tc & ::= & c \mid \alpha \mid tc \to tc \mid tc \times tc \\ \ell & ::= & \{s_1, \ldots, s_n\} \\ s & ::= & \delta \mid A \\ H & ::= & H \texttt{ at } A \mid [x_1 : \sigma_1, \ldots, x_n : \sigma_n] \\ C & ::= & \{s_1 \rhd s_1', \ldots, s_n \rhd s_n'\} \end{array}$$

H is the spatial typing environments. H `at` A denotes a located environment, i.e., a typing environment from which every spatial type will be forced to represent a value entirely located on A.

$$\text{(IMM)}\quad \hat{R} \overset{\ell}{\Vdash} i \xrightarrow{i \texttt{ at } s} i \qquad \text{(COMM)}\ \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{dv \texttt{ at } s} e' \qquad (s,s') \in \mathcal{L}}{\hat{R} \overset{\ell \cup \{s'\}}{\Vdash} e \xrightarrow{dv[s'/s] \texttt{ at } s'} e'}$$

$$\text{(INST)}\quad \hat{R}, [\hat{v}/x] \overset{\mathrm{loc}(\hat{v})}{\Vdash} x \xrightarrow{\hat{v}} x \qquad \text{(FBY)}\ \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} e_1 \texttt{ fby } e_2 \xrightarrow{\hat{v}_1} |\hat{v}_2| \texttt{ fby } e_2'}$$

$$\text{(OP)}\ \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{i_1 \texttt{ at } A} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{i_2 \texttt{ at } A} e_2' \qquad i = \mathrm{op}(i_1, i_2)}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} \mathrm{op}(e_1, e_2) \xrightarrow{i \texttt{ at } A} \mathrm{op}(e_1', e_2')}$$

$$\text{(PAIR)}\ \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{dv_1 \texttt{ at } s_1} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{dv_2 \texttt{ at } s_2} e_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} (e_1, e_2) \xrightarrow{(dv_1 \texttt{ at } s_1, dv_2 \texttt{ at } s_2) \texttt{ at } s_1 \sqcup s_2} (e_1', e_2')}$$

$$\text{(AT)}\ \frac{\hat{R} \overset{\{s\}}{\Vdash} e \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\{s\}}{\Vdash} e \texttt{ at } s \xrightarrow{\hat{v}} e' \texttt{ at } s}$$

$$\text{(DEF)}\ \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{(\hat{v}_1, \ldots, \hat{v}_n)} e'}{\hat{R} \overset{\ell}{\Vdash} (x_1, \ldots, x_n) = e \xrightarrow{[\hat{v}_i/x_i]} (x_1, \ldots, x_n) = e'}$$

$$\text{(APP)}\ \frac{\hat{R}(f) = \Lambda \delta_1, \ldots, \delta_n . \lambda p . e \texttt{ with } D \texttt{ at } s \qquad \{s_1, \ldots, s_n\} \subseteq \mathcal{S} \qquad \hat{R} \overset{\ell}{\Vdash} p' = e[\vec{s}/\vec{\delta}] \texttt{ and } p = e' \texttt{ and } D[\vec{s}/\vec{\delta}] \xrightarrow{\hat{R}'} D'}{\hat{R} \overset{\ell}{\Vdash} p' = f(e') \xrightarrow{\hat{R}'} D'}$$

$$\text{(AND)}\ \frac{\hat{R}, \hat{R}_2 \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D_1' \qquad \hat{R}, \hat{R}_1 \overset{\ell_2}{\Vdash} D_2 \xrightarrow{\hat{R}_2} D_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} D_1 \texttt{ and } D_2 \xrightarrow{\hat{R}_1, \hat{R}_2} D_1' \texttt{ and } D_2'}$$

$$\text{(IF-1)}\ \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{true at } s} e' \qquad \hat{R} \overset{\ell'}{\Vdash} D_1 \xrightarrow{\hat{R}'} D_1' \qquad \forall s' \in \ell', s \triangleright s'}{\hat{R} \overset{\ell \cup \ell'}{\Vdash} \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{\hat{R}'} \texttt{if } e' \texttt{ then } D_1' \texttt{ else } D_2}$$

$$\text{(IF-2)}\ \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{false at } s} e' \qquad \hat{R} \overset{\ell'}{\Vdash} D_2 \xrightarrow{\hat{R}'} D_2' \qquad \forall s' \in \ell', s \triangleright s'}{\hat{R} \overset{\ell \cup \ell'}{\Vdash} \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{\hat{R}'} \texttt{if } e' \texttt{ then } D_1 \texttt{ else } D_2'}$$

Figure 3. Distributed synchronous semantics.

We distinguish spatial type schemes (σ), which can be quantified, from simple spatial types (t). A set of constraints C can be associated to quantification of location variables ($\forall \delta_1, \ldots, \delta_n : C.t$). We note $\forall \delta_1, \ldots, \delta_n . t$ the scheme $\forall \delta_1, \ldots, \delta_n : \emptyset . t$. A simple spatial type can be either a node type ($t \multimap\!\langle \ell \rangle\!\rightarrow t$), a pair type ($t \times t$), or a located type ($tc$ `at` s). A located type can be either a stream type (c, such as boolean, integer, etc.), a type variable (α), a local function ($tc \rightarrow tc$), or a local pair type ($tc \times tc$). ℓ denotes sets of locations. A location is either a location variable δ, or a location A.

C is a set of constraints between locations. A constraint $s_1 \triangleright s_2$ means that either $s_1 = s_2$, or there exists a communication link from s_1 to s_2. Conversely, a declaration of communications links $\mathcal{L}$ leads to the set of constraints $\mathrm{constr}(\mathcal{L}) = \{s \triangleright s' | (s,s') \in \mathcal{L}\}$.

A value of spatial type tc `at` s is a value located on s. A value of spatial type $t_1 \multimap\!\langle \ell \rangle\!\rightarrow t_2$ is a node whose input is of spatial type t_1, whose output is of spatial type t_2, and whose computation involves the set of locations ℓ. The following equalities stand:

$$(tc_1 \times tc_2) \texttt{ at } s \;=\; (tc_1 \texttt{ at } s) \times (tc_2 \texttt{ at } s)$$
$$(tc_1 \rightarrow tc_2) \texttt{ at } s \;=\; (tc_1 \texttt{ at } s) \multimap\!\langle \{s\} \rangle\!\rightarrow (tc_2 \texttt{ at } s)$$

$$\frac{t_1 = t_1' \quad t_2 = t_2'}{(t_1 \times t_2) = (t_1' \times t_2')} \qquad \frac{t_1 = t_1' \quad t_2 = t_2'}{t_1 \multimap\!\langle \ell \rangle\!\rightarrow t_2 = t_1' \multimap\!\langle \ell \rangle\!\rightarrow t_2'}$$

The instantiation mechanism ensures the localization of a type instantiated from a located environment:

$$(t[tc_1/\alpha_1, \ldots, tc_n/\alpha_n, s/\delta], C[s_1/\delta_1, \ldots, s_m/\delta_m]) \leq \forall \alpha_1 \ldots \alpha_n \forall \delta_1 \ldots \delta_m : C.t$$

$$(tc \texttt{ at } s, C) \leq (H \texttt{ at } s)(x) \Leftrightarrow (tc \texttt{ at } s, C) \leq H(x)$$

We note respectively $\mathrm{FLV}(t)$ and $\mathrm{FTV}(t)$ the set of free location variables and free type variables of the type t. FLV and FTV are straightforwardly extended to typing environments.

A set of constraints C is *compatible* with a set of communication links $\mathcal{L}$, noted $\mathcal{L} \models C$, iff $s \triangleright s' \in C \wedge s \neq s' \Rightarrow (s,s') \in \mathcal{L}$.

Before presenting our spatial type system, we introduce the following notations:

- For a program P, the notation $\vdash P : t$ means that the program P is of spatial type t.
- For declarations (resp. expressions), the notation $H|G \vdash D : H'/\ell$ (resp. $H|G \vdash e : t/\ell$) means that, in the spatial type environment H and the architecture graph G, the declaration D (resp. the expression e) defines a new environment H' (resp. is of spatial type t), and its computation involves the set of locations ℓ.

The function $\mathrm{locations}(\cdot)$ gives the set of locations involved in the spatial type given as argument. It is defined as:

$$\begin{array}{ll} \mathrm{locations}(t_1 \times t_2) & = \mathrm{locations}(t_1) \cup \mathrm{locations}(t_2) \\ \mathrm{locations}(t_1 \multimap\!\langle \ell \rangle\!\rightarrow t_2) & = \ell \\ \mathrm{locations}(tc \texttt{ at } s) & = \{s\} \end{array}$$

The top-level declaration of a program is typed from the initial environment H_0, defined as:

$$H_0 = \left[\begin{array}{l} \cdot \texttt{ fby } \cdot : \forall \alpha . \forall \delta . \alpha \texttt{ at } \delta \times \alpha \texttt{ at } \delta \multimap\!\langle \{\delta\} \rangle\!\rightarrow \alpha \texttt{ at } \delta, \\ (+) : \forall \delta . c \texttt{ at } \delta \times c \texttt{ at } \delta \multimap\!\langle \{\delta\} \rangle\!\rightarrow c \texttt{ at } \delta, \ldots \end{array} \right]$$

Our spatial type system is formally defined by the inference rules shown in Figure 4. Typing a program involves building an architecture graph from the architecture description, and then using it to type the nodes and the main declarations (rule PROG).

An immediate value can be used on any location (rule IMM). Type schemes can be instantiated (rule INST). Typing a pair involves stating that this pair has to be evaluated on the union of the sets of locations on which each member of the pair has to be evaluated (rule PAIR). Typing a located expression (e `at` A) involves building a located typing environment (rule AT). Communications are expressed as subtyping (rule COMM).

The spatial type of a node consists of the spatial types of its inputs, the computed expression, and the set of locations involved in this computation (rule NODE). The type of a node is generalized w.r.t. the set of locations and links introduced by this architecture.

Typing an equation $x = e$ involves building a singleton typing environment (rule DEF). Rule APP states that an application must be evaluated on the union of the set of locations where the node f and its argument e must be evaluated, and the set of locations ℓ_1 involved in the computation of the node f. Parallel declarations involve, for their computations, the union of the sets and of locations involved in the computation of their compounds (rule AND). Finally, typing an `if`/`then`/`else` declaration involves locating the condition expression on a location s, and adding constraints that every location involved in declarations D_1 and D_2 must be accessible from s (rule IF).

$$\text{(PROG)}\ \frac{\langle\emptyset,\emptyset\rangle \vdash \mathcal{A} : G \qquad H_0|G \vdash d : H/\ell \qquad H, H_1|G \vdash D : H_1/\ell'}{\vdash \mathcal{A}; d; D : H_1}$$

$$\text{(IMM)}\ H|G \vdash i : c\ \texttt{at}\ s/\{s\} \qquad \text{(INST)}\ \frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash x : t/\text{locations}(t)}$$

$$\text{(PAIR)}\ \frac{H|G \vdash e_1 : t_1/\ell_1 \qquad H|G \vdash e_2 : t_2/\ell_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2}$$

$$\text{(DEF)}\ \frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell}$$

$$\text{(AT)}\ \frac{H\ \texttt{at}\ s|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : t/\ell \qquad s \in \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e\ \texttt{at}\ s : t/\ell}$$

$$\text{(COMM)}\ \frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s/\ell \qquad \mathcal{L} \models s \triangleright s'}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s'/\ell \cup \{s'\}}$$

$$\text{(NODE)}\ \frac{\begin{array}{c} H, x_i : t_i, H_1|\langle\mathcal{S}',\mathcal{L}'\rangle \vdash D : H_1/\ell_1 \\ H, x_i : t_i, H_1|\langle\mathcal{S}',\mathcal{L}'\rangle \vdash e : t/\ell_2 \qquad \mathcal{S}' = \mathcal{S} \cup \{\delta_1, \ldots, \delta_p\} \\ \mathcal{L}' \subseteq \mathcal{L} \cup (\{\delta_1, \ldots, \delta_p\} \times \mathcal{S}) \cup (\mathcal{S} \times \{\delta_1, \ldots, \delta_p\}) \\ \{\alpha_1, \ldots, \alpha_m\} = \text{FTV}(t) - \text{FTV}(H) \qquad C = \text{constr}(\mathcal{L}' \setminus \mathcal{L}) \\ \sigma = \forall\alpha_1, \ldots, \alpha_m.\forall\delta_1, \ldots, \delta_p : C.(t_1 \times \ldots \times t_n) \multimap\langle\ell_1 \cup \ell_2\rangle\mapsto t \end{array}}{H|G \vdash \texttt{node}\ f[\delta_1, \ldots, \delta_p](x_1, \ldots, x_n) = e\ \texttt{with}\ D : [\sigma/f]/\ell_1 \cup \ell_2}$$

$$\text{(APP)}\ \frac{H|G \vdash f : t \multimap\langle\ell_1\rangle\mapsto (t'_1 \times \ldots \times t'_n)/\ell_2 \qquad H|G \vdash e : t/\ell_3}{H|G \vdash (x_1, \ldots, x_n) = f(e) : [t'_1/x_1, \ldots, t'_n/x_n]/\ell_1 \cup \ell_2 \cup \ell_3}$$

$$\text{(AND)}\ \frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H|G \vdash D_2 : H_2/\ell_2}{H|G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2}$$

$$\text{(IF)}\ \frac{\begin{array}{c} H|G \vdash e : c\ \texttt{at}\ s/\ell \qquad H|G \vdash D_1 : H'/\ell_1 \\ H|G \vdash D_2 : H'/\ell_2 \qquad \mathcal{L} \models \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \end{array}}{H|G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2}$$

Figure 4. Spatial type system.

We denote by $\hat{v} : t$ the fact that the distributed value $\hat{v}$ has spatial type t:

$$lv\ \texttt{at}\ s : c\ \texttt{at}\ s \qquad \frac{\hat{v}_1 : t_1 \qquad \hat{v}_2 : t_2}{(\hat{v}_1, \hat{v}_2) : t_1 \times t_2}$$

We denote by $\hat{R} : H$ the type compatibility between $\hat{R}$ and H:

$$\hat{R} : H \Leftrightarrow \forall x \in \text{dom}(\hat{R}), x \in \text{dom}(H) \\ \wedge\, \exists (t, C) \text{ s.t. } (t, C) \leq H(x) \wedge \hat{R}(x) : t$$

Theorem 1 states that if a program reacts with the centralized semantics, and is accepted by our type system, then there exists a spatialized execution such that the distributed values of this execution are equal to the centralized ones. The types are preserved by this spatial execution. The proof is omitted for lack of space.

Theorem 1 (Soundness)**.** *For all D,D',H,H',R,R',G, if $H|G \vdash D : H'/\ell$ and $R \vdash D \xrightarrow{R'} D'$, then there exists $\hat{R}, \hat{R}'$ such that $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$, $\hat{R} : H$, $\hat{R}' : H'$, $|\hat{R}| = R$ and $|\hat{R}'| = R'$.*

4. Distribution

4.1 Principle

Once programs have been typed, every expression is annotated with a location that specifies where it has to be computed. Communications are inserted when a value is produced at a location and used at another. From this typed program, the compiler produces several new programs — one for every location s — erasing the code that is not necessary at this location s. The run-time we have chosen is a classical one for globally asynchronous locally synchronous (GALS) systems: communications are done through FIFOs.

We show below the result of the projection of the node `f` of Section 2.2 on `A` and `B`, noted respectively `f_A` and `f_B`. The distribution of this node will involve adding a communication between these two computations. This communication will take the form of an additional output on `f_A` (named here `c_y`, holding the value `y` computed on `A` and used on `B`), together with an additional input on `f_B`. Original inputs and outputs are not suppressed: $\perp$denotes an irrelevant value which will not be used on the current location. It is used here to replace the output `z`, whose computation is suppressed at `A`.

```
node f_A(x) = (⊥,c_y) with
   c_y = g(x)

node f_B(x,c_y) = z with
   z = h(c_y)
```

The semantically equivalent distributed system is then obtained by connecting the input and output `c_y`, holding the communicated value `y`. The program below shows the distributed execution, using a FIFO materialized by `send`/`receive` primitives, of the result of the projection of the program `y = f(x)`.

```
(y_A,c_y) = f_A(x);   |  receive(c_y);
send(c_y)             |  y_B = f_B(⊥,c_y)
```

4.2 Example

The result of the projection of the two nodes of the Section 2.5 on the location `DSP` is given on Figure 5. The projection of the `channel` node shows that the node applications of `filter` and `crc` have been removed, and that a new input `c1` (holding the value of `f`) and a new output `c2` (holding the value of `d`) have been added. This implies the addition, on the projection of the `multichannel_sdr` node, of two new inputs (`c2` and `c3`) and two new outputs (`c4` and `c5`), one for each `channel` instance. The new input `c1` of the projected `multichannel_sdr` node holds the value `c`.

```
node channel(filter,demod,crc,x,c1) = (⊥,c2) with
      d = demod(c1)
  and c2 = d

node multichannel_sdr(x,s,c1,c2,c3) = (⊥,c4,c5)
  if c1 then
    (y,c4) = channel(⊥,gmsk,⊥,⊥,c2)
  else
    (y,c5) = channel(⊥,qpsk,⊥,⊥,c3)
```

Figure 5. Result of the projection on `DSP`

4.3 Projection

We will now define a type-directed operation of *projection of an expression on a location A*. This operation is defined separately, as it has to be performed on an already annotated program: links between values of each projected program are defined by the channels inferred by the type system. The projection will use a refined version of the type system, allowing the inference of communication channels.

A channel is a location pair associated with a name, noted $A_1 \overset{c}{\mapsto} A_2$: c is the name of the channel, A_1 its source location, and A_2 its destination location. The set of channel names is ordered by $<$, so as to keep consistency of inputs and outputs added, from the

node definition to node instances. T denotes sequences of channels. The concatenation of two sequences of channels, noted T_1, T_2, is defined iff channel names in T_1 and T_2 are disjoint. We denote by ϵ the empty sequence.

We note $\mathrm{dom}(T)$ the set of channel names of T. $T' \cong T$ means that the sequences T and T' are equal, modulo channel renaming. This renaming allows the multiple instanciations of node comprising communications.

$T' \cong T \Leftrightarrow T' = T[c'_1/c_1,...,c'_n/c_n]$ where $\{c_1,...,c_n\} = \mathrm{dom}(T)$

The projection of a declaration D on a location A is noted $H|G \vdash D : H'/\ell/T \overset{A}{\Longrightarrow} D'$, and results in a new declaration D', containing only the computations to be performed on A. The projection of an expression e, of spatial type t, on a location A, is noted $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} e'/D$, and results in a new expression e', as well as a declaration D, containing channels outputs to be defined. A channel named c in an environment channel will be introduced as the variable c as input or output of the target program, c assumed to be of different name space than other variables of the source program.

We denote by ϵ the empty declaration, and by $\perp$ a value which will never be used (i.e., void). For any declaration D, D `and` $\epsilon = \epsilon$ `and` $D = D$. For any expression e, we have $(\perp\ e) = (e\ \perp) = \perp$.

Also, we note $T \uparrow A$ (resp. $T \downarrow A$) the set of channels with origin (resp. destination) A:

$$\begin{cases} \emptyset \uparrow A = \emptyset \\ ([A_1 \overset{c}{\mapsto} A_2], T) \uparrow A = \begin{cases} [A_1 \overset{c}{\mapsto} A_2], (T \uparrow A) & \text{if } A_1 = A \\ (T \uparrow A) & \text{else,} \end{cases} \end{cases}$$

$$\begin{cases} \emptyset \downarrow A = \emptyset \\ ([A_1 \overset{c}{\mapsto} A_2], T) \downarrow A = \begin{cases} [A_1 \overset{c}{\mapsto} A], (T \downarrow A) & \text{if } A_2 = A \\ (T \downarrow A) & \text{else.} \end{cases} \end{cases}$$

The projection rules are given in Figures 6 and 7. Channels are used at communication points. If an expression e is sent from A to A' through the channel $A \overset{c}{\mapsto} A'$, then:

- for the projection on A, the communication involves sending a value: the resulting expression is void, and we add the definition of the channel c as the result of the projection of e on A (rule COMM-P-FROM);
- for the projection on A', the communication involves receiving a value: the resulting expression is the channel holding this value (rule COMM-P-TO).

Finally, if A does not appear in the set of locations involved in its computation, then the expression can be suppressed on A (rule SUPPR-P). Projections of a pair consist in the projection of its compounds (rule PAIR-P).

The projection of a located declaration and parallel declarations involves the projection of its compound (rules AT-P and AND-P). The projections of applications and node definitions involve adding to the inputs and outputs of the node, the channels used by this node (rules APP-P and NODE-P). Nodes with local architecture are assumed to be inlined. The relevance of the name order appears here, as the order of the added inputs and outputs must be consistent with every instances of these nodes, and for every projection.

Projection of a conditional is divided in two rules: one for the projection on a location where the conditional expression is computed: this first rule shows the definition of every channel needed to send this value to other locations where the conditional will be evaluated (rule IF-P-FROM); and one for the projection on a location where the conditional expression has to be received: this expression is then replaced by the name of the channel holding this value (rule IF-P-TO).

(IMM-P)
$$H|G \vdash i : c\ \texttt{at}\ s/\{s\}/\epsilon \overset{A}{\Longrightarrow} i/\epsilon$$

(INST-P)
$$\frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|G \vdash x : t/\ell/\epsilon \overset{A}{\Longrightarrow} x_A/\epsilon}$$

(AT-P)
$$\frac{H\ \texttt{at}\ A|G \vdash D : H'/\ell/T \overset{A}{\Longrightarrow} D'}{H|G \vdash D\ \texttt{at}\ A : H'/\ell/T \overset{A}{\Longrightarrow} D'}$$

(SUPPR-P)
$$\frac{A \notin \ell}{H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} \perp/\epsilon}$$

(COMM-P-FROM)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ A/\ell/T \overset{A}{\Longrightarrow} e'/D \qquad \mathcal{L} \models A \triangleright s'}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s'/\ell \cup \{s'\}/T, [A \overset{n}{\mapsto} s'] \overset{A}{\Longrightarrow} \perp/D\ \texttt{and}\ c_n = e'}$$

(COMM-P-TO)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s/\ell/T \overset{A'}{\Longrightarrow} e'/D \qquad \mathcal{L} \models s \triangleright A'}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ A'/\ell \cup \{A'\}/T, [s \overset{n}{\mapsto} A'] \overset{A'}{\Longrightarrow} c_n/D}$$

(PAIR-P)
$$\frac{H|G \vdash e_1 : t_1/\ell_1/T_1 \overset{A}{\Longrightarrow} e'_1/D_1 \quad H|G \vdash e_2 : t_2/\ell_2/T_2 \overset{A}{\Longrightarrow} e'_2/D_2}{H|G \vdash e_1, e_2 : t_1 \times t_2/\ell_1 \cup \ell_2/T_1, T_2 \overset{A}{\Longrightarrow} e'_1, e'_2/D_1\ \texttt{and}\ D_2}$$

(DEF-P)
$$\frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell/T \overset{A}{\Longrightarrow} e'/D}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T \overset{A}{\Longrightarrow} (x_{1A}, \ldots, x_{nA}) = e'\ \texttt{and}\ D}$$

(APP-P)
$$\frac{\begin{array}{c} H|G \vdash f : t \multimap (\ell_1/T_1) \mapsto (t'_1 \times \ldots \times t'_n)/\ell_2/T_2 \overset{A}{\Longrightarrow} f'/D_1 \\ H|G \vdash e : t/\ell_3/T_3 \overset{A}{\Longrightarrow} e'/D_2 \qquad T'_1 \cong T_1 \\ T'_1 \uparrow A = [A \overset{c_1}{\mapsto} A_1, \ldots, A \overset{c_m}{\mapsto} A_m] \\ T'_1 \downarrow A = [A'_1 \overset{c'_1}{\mapsto} A, \ldots, A'_p \overset{c'_p}{\mapsto} A] \end{array}}{\begin{array}{c} H|G \vdash (x_1, \ldots, x_n) = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T'_1, T_2, T_3 \\ \overset{A}{\Longrightarrow} (x_{1A}, \ldots, x_{nA}, c_1, \ldots, c_m) = f'(e', c'_1, \ldots, c'_p)\ \texttt{and}\ D_1\ \texttt{and}\ D_2 \end{array}}$$

(AND-P)
$$\frac{H|G \vdash D_1 : H_1/\ell_1/T_1 \overset{A}{\Longrightarrow} D'_1 \qquad H|G \vdash D_2 : H_2/\ell_2/T_2 \overset{A}{\Longrightarrow} D'_2}{H|G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2 \overset{A}{\Longrightarrow} D'_1\ \texttt{and}\ D'_2}$$

Figure 6. Rules for the projection operation (I).

The global meaning of a distributed program is then defined by the parallelization of its projected declarations.

We note $\mathcal{S} = \{A_1, \ldots, A_n\}$ the set of defined constant locations where the source declarations are projected. The global meaning of a declaration D, projected on the locations $\mathcal{S}$, is defined by:

$$D_1\ \texttt{and}\ \ldots\ \texttt{and}\ D_n \text{ where } \forall i, H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$$

In order to relate a target program with its source, we define a relation on values, denoted $\cdot \preccurlyeq^{\cdot}_{\cdot} \cdot$, such that $v' \preccurlyeq^A_t v$ means that the value v', emitted from an expression of type t, represents the value v at the location A. We have:

$$\frac{v' = v}{v' \preccurlyeq^A_{t\ \texttt{at}\ A} v} \qquad \frac{A \neq A'}{\perp \preccurlyeq^A_{t\ \texttt{at}\ A'} v} \qquad \frac{v'_1 \preccurlyeq^A_{t_1} v_1 \qquad v'_2 \preccurlyeq^A_{t_2} v_2}{(v'_1, v'_2) \preccurlyeq^A_{t_1 \times t_2} (v_1, v_2)}$$

We can then relate two reaction environments R and R_p w.r.t. a typing environment H:

$$R \preccurlyeq_H R_p \text{ iff } \forall x \in \mathrm{dom}(R), \forall A \in \mathcal{S}, R(x) \preccurlyeq^A_{H(x)} R_p(x_A)$$

Theorem 2 states that the projection operation is correct, i.e., the projection of a source program D into D_i (for every location A_i) defines a new target program D_1 `and` $\ldots$ `and` D_n, which is semantically equivalent, taking into account spatial types' values, with the source declaration D. The proof is omitted for lack of space.

Theorem 2 (Soundness of the declarations projection)**.** *For all H, H', D, D', ℓ, T, D_i, R, R', if $R \vdash D \xrightarrow{R'} D'$, $H|G \vdash D : H'/\ell/T$ and $\forall i, H|G \vdash D : H'/\ell/T \stackrel{A_i}{\Longrightarrow} D_i$, then there exists R_p, R'_p, D_p, D'_p such that $D_p = D_1$ `and` $\ldots$ `and` D_n, $R \preccurlyeq_H R_p$, $R_p \vdash D_p \xrightarrow{R'_p} D'_p$, and $R' \preccurlyeq_{H'} R'_p$.*

5. Discussion

5.1 Implementation

The spatial type system presented in Section 3.4 relies on a subtyping mechanism. It corresponds to the case where communications can occur *anywhere* in the code. This situation raises two problems. Firstly, the implementation of type systems with subtyping mechanism is costly: usual algorithms rely on the systematic application of the subtyping rule. Secondly, this choice leads to a situation where the programmer has no control over where the communications can occur. These problems, though orthogonal, can be addressed together: giving some control to the programmer means restricting the points where subtyping can be applied. We refine our type system in order to address these two problems.

We restrain communicated values to be variables introduced by equations ($x = e$). Thus, in the program of Section 2.2, only `y` and `z` can be communicated from one location to another. Then, we can use a generalization mechanism to infer communication constraints, instead of inferring them by subtyping. This corresponds to restricting the subtyping mechanism to instantiation points.

This refined type system, as well as the projection operation presented in Section 4, have been implemented in the Lucid Synchrone compiler [1]. The independence of the method w.r.t. other analysis allowed this implementation to be quite modular, implying very few modifications within the rest of the compiler. The implementation consists to generate distribution constraints, which are resolved modularly for each node. We implemented a naive resolution algorithm, which can easily be replaced by any existing distribution algorithm, taking into account efficiency issues (for example, the AAA method of SynDex [18]). This implementation has been tested on the software-defined radio example, which is in its actual size composed of 150 lines of code, spread out into 25 Lucid Synchrone nodes. We have tested our automatic distribution algorithm on a benchmark suite composed of programs of few hundred of lines from the current Lucid Synchrone release. This point is a good indication of the flexibility of the extension proposed, allowing the distribution of programs not primarily designed with distribution in mind. The scalability of this method is a direct outcome of the use of a type system aiming at modular distribution; it has been also applied on an ad hoc example composed of 10 Lucid Synchrone nodes, comprising each 60 equations.

5.2 Related Work

Various solutions have emerged in order to use synchronous languages for the design of distributed systems. Some of them operate on a compiled model of the program, by "coloring" atomic instructions with localization information, inferred from input and output locations [8]. The whole program is first *compiled* into an intermediate *sequential* format, on which the distribution is then applied. This format consists of a sequence of atomic instructions, representing one computation step of new values carried on each stream. Then, the distribution involves placing each instruction onto one or several locations, taking into account the consistency of the control flow on each location. Another approach is to directly annotate the source program with *locations*, so as to define the localization of each variable of the program: the distribution is then performed with regard to these annotations. This approach has been applied to Signal [3] as well as Lustre [7]. In both cases, the soundness of the distribution algorithm has been proved [6, 14], meaning that the combined behavior of the distributed fragments has the same functional and temporal semantics as the initial centralized program. The originality of our method resides in the fact that we use a spatial type system to check the consistency of the distribution specifications inserted by the programmer, and that we perform *modular* distribution, allowing the expression of higher-order features, applied for instance to dynamic reconfiguration of nodes by application of other nodes as inputs. Since a Kahn semantics [11] can be given to our language, the semantic equivalence between the source program and the synchronous product of the fragments resulting from the projection is sufficient to describe an asynchronous distribution. While the language presented has higher-order features, this method can easily be applied to other language with comparable semantics such as Lustre. In contrast, more general frameworks such as Signal cannot be addressed here for this reason.

Several approaches have been considered to solve the problem of data consistency of distributed programs. A translation operation is presented in [16], as well as an effect type system, in order to automatically obtain a multi-tier application from an annotated source program. Our proposition differs by the fact that our type system is not only a specification, but also consists in what the authors called "location analysis", thus allowing us to perform this analysis in a modular way, and on a program comprising higher-order features. This last approach, as well as our projection operation, can be compared with slicing methods [20], as they consists in extracting specific parts of a global program. Type systems have been used to ensure memory consistency [19], or for pointer analysis within a distributed architecture [12]. The ACUTE language [17] is an extension of OCAML with typed marshalling. Communication channels between two ACUTE programs can also be considered as typed, as the type of marshalled and unmarshalled data are dynamically verified, at execution time. The consistency considered is between separately-built programs, whereas our approach is to consider the programming of a distributed system as one global program, allowing global static verifications. Our approach can be compared with automatic partitioning: the J-Orchestra system [13] allows the user to assign network sites to classes of Java programs. Then, this automatic partitioning system transform the initial program into a distributed one, taking into account distant or direct references. This last approach differs with our by the fact that we integrate the distribution constructs within our language, which allows relying distribution on semantical basis. Finally, OZ and its distributed extension [4] proposes a way to separate the functionality of a distributed application and its distribution structure, by allowing the programmer to give a different distributed semantics to every different object of a program. These two languages aims at loosely coupled distributed systems without architecture constraints, whereas our approach concerns strongly coupled architecture, and we aim to ensure the consistency of the distribution w.r.t. one architecture, given the communication constraints.

5.3 Conclusion and Future Work

We have proposed a spatial type system to solve the problem of automatic distribution of dataflow programs. It is based on a core dataflow language, which we have extended with distribution primitives to allow the programmer to specify, on one hand his/her target distributed architecture, and on the other hand where some nodes and/or variables are to be located. The underlying philosophy is

$$\text{(NODE-P)}\ \frac{\begin{array}{c} H, x_i : t_i, H_1 | G \vdash D : H_1/\ell_1/T_1 \overset{A}{\Longrightarrow} D' \qquad H, x_i : t_i, H_1 | G \vdash e : t/\ell_2/T_2 \overset{A}{\Longrightarrow} e'/D_e \\ \{\alpha_1, \ldots, \alpha_m\} = \mathrm{FTV}(t) - \mathrm{FTV}(H) \qquad \sigma = \forall \alpha_1, \ldots, \alpha_m . (t_1 \times \ldots \times t_n) \xrightarrow{(\ell_1 \cup \ell_2/T_1, T_2)} t \\ T_1, T_2 \uparrow A = [A \overset{c_1}{\mapsto} A_1, \ldots, A \overset{c_p}{\mapsto} A_p] \qquad T_1, T_2 \downarrow A = [A'_1 \overset{c'_1}{\mapsto} A, \ldots, A'_q \overset{c'_q}{\mapsto} A] \end{array}}{\begin{array}{l} H | G \vdash \texttt{node}\ f(x_1, \ldots, x_n) = e\ \texttt{with}\ D : [\sigma/f]/\ell_1 \cup \ell_2/\emptyset \\ \overset{A}{\Longrightarrow} \texttt{node}\ f_A(x_{1A}, \ldots, x_{nA}, c'_1, \ldots, c'_q) = (e', c_1, \ldots, c_p)\ \texttt{with}\ D'\ \texttt{and}\ D_e \end{array}}$$

$$\text{(IF-P-FROM)}\ \frac{\begin{array}{c} H | G \vdash e : c\ \texttt{at}\ s/\ell/T \overset{A}{\Longrightarrow} e'/D \qquad H | G \vdash D_1 : H'/\ell_1/T_1 \overset{A}{\Longrightarrow} D'_1 \qquad H | G \vdash D_2 : H'/\ell_2/T_2 \overset{A}{\Longrightarrow} D'_2 \\ C = \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \qquad \mathcal{L} \models C \qquad T' = \mathrm{channels}(C) \qquad T' \uparrow A = [A \overset{c_1}{\mapsto} A_1, \ldots, A \overset{c_n}{\mapsto} A_n] \qquad x \notin \mathrm{dom}(H) \end{array}}{\begin{array}{l} H | G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, \mathrm{channels}(C) \\ \overset{A}{\Longrightarrow} x = e'\ \texttt{and}\ c_1 = x\ \texttt{and}\ \ldots\ \texttt{and}\ c_n = x\ \texttt{and}\ \texttt{if}\ x\ \texttt{then}\ D'_1\ \texttt{else}\ D'_2 \end{array}}$$

$$\text{(IF-P-TO)}\ \frac{\begin{array}{c} H | G \vdash e : c\ \texttt{at}\ s/\ell/T \overset{A}{\Longrightarrow} e'/D \qquad H | G \vdash D_1 : H'/\ell_1/T_1 \overset{A}{\Longrightarrow} D'_1 \\ H | G \vdash D_2 : H'/\ell_2/T_2 \overset{A}{\Longrightarrow} D'_2 \qquad C = \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \qquad \mathcal{L} \models C \qquad T' = \mathrm{channels}(C) \qquad T' \downarrow A = [A' \overset{c}{\mapsto} A] \end{array}}{H | G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, \mathrm{channels}(C) \overset{A}{\Longrightarrow} \texttt{if}\ c\ \texttt{then}\ D'_1\ \texttt{else}\ D'_2}$$

Figure 7. Rules for the projection operation (II).

the functional distribution, meaning that some functionalities of the program must be computed at some precise location because they require some specific sensors, actuators, and/or computing resources that are available only at this location. In this context, we use type inference to decide at which location each node must be computed, and at which points in the program communication primitives must be inserted. The compilation of a correctly spatially typed program produces one program for each computing location specified by the programmer. We use abstract communication channels to exchange a value between two locations and to synchronize them. Compiling each program and linking with a dedicated library implementing those communication channels then gives one binary code for each location. The refined version of the type system, as well as the operation projection, has been implemented in a synchronous dataflow compiler [1].

Future work mainly involves allowing the description of more complex architectures, with hierarchical locations or communication masking (e.g., MPSoCs): our proposal of architecture constraints is not sufficient to catch the complexity of actual architecture of distributed embedded systems. Yet, our current constraints show the interest of a type system for checking the consistency of a distributed program w.r.t. such constraints.

References

[1] Lucid synchrone v3. `www.lri.fr/~pouzet/lucid-synchrone`.

[2] Architecture analysis & design language (AADL). SAE Standard (AS5506), Nov. 2004.

[3] P. Aubry and P. Le Guernic. On the desynchronization of synchronous applications. In *11th International Conference on Systems Engineering, ICSE'96*, Las Vegas, USA, June 1996.

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1):64–83, Jan. 2003.

[6] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automated Systems*, 31(3):503–524, 1997.

[7] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03*, San Diego, USA, June 2003. ACM.

[8] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3):416–427, May 1999.

[9] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[10] F. Jondral. Software-defined radio — basics and evolution to cognitive radio. *EURASIP Journal on Wireless Communications and Networking*, 3:275–283, 2005.

[11] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475, New York, NY, 1974. North-Holland.

[12] B. Liblit and A. Aiken. Type systems for distributed data structures. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 199–213, New York, NY, USA, 2000. ACM Press.

[13] N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning: A promising approach to prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 2004.

[14] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de Signal*. Phd thesis, University of Rennes I, Rennes, France, Jan. 1993.

[15] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[16] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[17] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of ICFP 2005: International Conference on Functional Programming (Tallinn)*, Sept. 2005.

[18] Y. Sorel. SynDEx: System-level cad software for optimizing distributed real-time embedded systems. *Journal ERCIM News*, 59:68–69, Oct. 2004.

[19] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[20] M. Ward and H. Zedan. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):7, 2007.